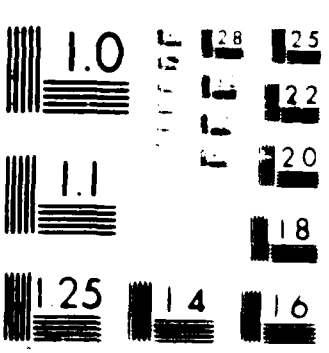
AD-A193 299 A TRANSPARENT COPROCESSOR FOR INTERPROCESSOR COMMUNICATION IN AN MIMD COMPUTER(U) WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE T J HOLMAN ET AL. 1/1

UNCLASSIFIED JAN 87 N00014-86-K-0264 F/G 12/6 NL


1.0
1.1
1.25
1.4
1.6
2.8
2.5
2.2
2.0
1.8

(4)

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Transparent Coprocessor for Interprocessor Communication in an MIMD Computer		5. TYPE OF REPORT & PERIOD COVERED: Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Thomas J. Holman, Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s): N00014-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Washington, Department of Computer Science, Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Arlington, VA 22227		12. REPORT DATE: January 1987
		13. NUMBER OF PAGES: 16
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel processing, MIMD, communication, coprocessor, message passing, communication delay

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents the design of a high performance interprocessor communication coprocessor for non-shared memory MIMD architectures. The design provides efficient interprocessor communication by relieving the computational processor of all communication related activities and by minimizing the overhead of packet assembly and disassembly. A multiprocessing scheme with zero process switch time allows this coprocessor to handle many communication ports with no additional overhead. Logical ports, which allow many computational processes to share the same physical port, are handled automatically by the coprocessor.

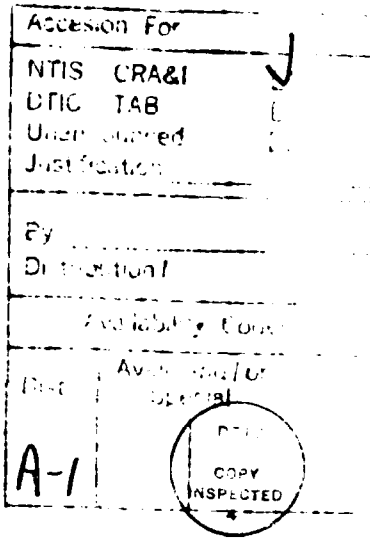
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A Transparent Coprocessor for Interprocessor Communication in an MIMD Computer

Thomas J. Holman
Lawrence Snyder
Computer Science Department, FR-35
University of Washington
Seattle, Washington 98195

January 1987



Abstract

This paper presents the design of a high performance interprocessor communication coprocessor for non-shared memory MIMD architectures. The design provides efficient interprocessor communication by relieving the computational processor of all communication related activities and by minimizing the overhead of packet assembly and disassembly. A multiprocessing scheme with zero process switch time allows this coprocessor to handle many communication ports with no additional overhead. Logical ports, which allow many computational processes to share the same physical port, are handled automatically by the coprocessor.

This work was supported in part by the Office of Naval Research under contract N00014-86-K-0264 and by the National Science Foundation under grant number DCR-8416878

1 Overview

It is not enough for parallel computers to apply multiple processors to a problem; parallelism should be exploited in every aspect of the computer organization. This "use parallelism everywhere" rule is especially relevent for interprocessor communication since it commonly happens that there is communication and computational work to be done at the same time. It follows then that organizing a processor element to have a coprocessor devoted exclusively to interprocessor communication is a sensible idea. However, if the interface between the computational processor and the communication coprocessor is not well thought out, the problem will not have been solved, but rather displaced by one level of indirection: The processor will now waste time communicating with the communication coprocessor! In this paper we present a design of a communication coprocessor suitable for non-shared memory MIMD architectures that provides a transparent interface to the computational processor.

2 Introduction

The performance of non-shared memory MIMD parallel computers is, in general, limited by the high cost of interprocessor communication. Further, in algorithms for this architecture, communication requirements are typically comparable to computation requirements, creating a high demand for a scarce resource. Three factors contribute to this "communications bottleneck". The first factor is the overhead incurred during the movement of data from a data structure to a port in the sender, and from a port to a data structure in the receiver. Specific sources of overhead for the sender include determining when the data is ready, moving the data from the data structure to the port, and preparing the data packet for transmission. For the receiver, the overhead comes from verifying and acknowledging the packet, extracting the data, moving the data to its destination data structure, and finally signalling data availability. We will call this the *staging overhead*.

The second factor involved in the communications bottleneck is the overhead of transmitting control information along with the data, which we call the *transmission overhead*. This includes protocol bits, packet headers, and fault detection/correction information. If a packet has m bits total and n bits of control information, then the transmission overhead is n/m. For example, in the Transputer [1], a packet has 11 bits, 3 of which are control bits, giving a transmission overhead of 0.27.

Finally, the third factor contributing to the communications bottleneck is the combination of the bandwidth of the communications port, called *external bandwidth* and the bandwidth of the processing node's internal path used for staging, called *internal band-*

width. This imposes a lower bound on the time to make a data transfer, which is achieved when both the staging and transmission overheads are zero. While this bound is not attainable for the systems of interest here, it is a useful measure of the efficiency of a communications architecture.

In this paper, we present the design of a communication coprocessor that reduces both staging overhead and transmission overhead. Overhead is reduced in this architecture by direct methods, which are described below, as well as by taking full advantage of the concurrency achieved by having a separate processor for communications operations. The methods used to interface the coprocessor to the main processor relieve the main processor of all communications related activities. Transmission overhead is minimized by a fault tolerant protocol streamlined for this parallel architecture. We view bandwidth enhancement as an independent, technology related problem, which is beyond the scope of this paper.

We begin in Section 3 with a presentation of the design goals that we consider important in reducing overhead. The communication coprocessor architecture motivated by these goals is presented in the three sections that follow: Section 4 describes the interface between the coprocessor and the main processor, the port interface is presented in Section 5, and Section 6 gives the details of the overall architecture uniting these interfaces.

3 Design Goals

Our basic goal is to minimize both staging and transmission overhead in non-shared memory MIMD architectures, sometimes called "ensemble architectures" [6], or microcomputer arrays [4]. These architectures typically consist of a large number of identical processing nodes linked together by a point-to-point communications graph. Each node is the combination of a serial processor, memory, and some number of communication ports with their associated hardware. Examples of this type of architecture are the Cosmic Cube [5] and the CHiP Computer [8].

Typically, the communication hardware in these MIMD architectures is a simple controller under command of the main processor, as in the Mosaic [2] element. A more sophisticated system, such as the Transputer, may include DMA capabilities. One of the drawbacks of these designs for handling communications is that the main processor is required to perform some or all of the communications related operations. Because there can be a substantial amount of communications activity, overall performance can be significantly reduced. A second drawback with these designs is that the memory and system bus are time multiplexed between computation and communication, rather than allowing these operations to proceed in parallel. Contention for these resources further reduces

performance. Both of these drawbacks manifest themselves as staging overhead for the communications.

The first drawback can be effectively eliminated by using a communication coprocessor to perform all communications related activities. The main processor will now have more time for computation. However, this does not necessarily isolate the main processor from communications related activities. If we continue to use a shared bus, as in the Caltech/JPL hypercube architectures [3,9], we will still have the second problem, i.e. there will be contention between the processors for bus cycles. Also, having two communicating processors introduces the additional problem of requiring some method of synchronization, which adds overhead. Our goal is to develop a design that eliminates both contention and synchronization overhead.

Another source of staging overhead is the movement of data to and from intermediate buffers. We would like to eliminate any excessive movement of data by reading and writing data directly to and from the destination data structure, that is, the data structure and the buffer are one and the same. When intermediate buffers separate from a computational data structure are required, they can be used, but we consider this to be the rare case.

Since we view the main processor as multitasking, it is possible to have more than one task using a particular physical port. The communication coprocessor must then be able to map the logical ports associated with a process to physical ports. When a packet arrives at a physical input port, it will be tagged with the number of the destination logical buffer. The coprocessor must interpret this number and send the data to the appropriate location in memory. Data written to a logical output buffer, i.e. to some location in memory, must in turn be sent out the appropriate physical port.

In addition, since the communication coprocessor must handle a number of physical ports, it to should have multiple processes. These processes link logical buffers to buffers in the physical ports. Because there are multiple ports each with one or more processes, polling by a process waiting for data or space can potentially take cycles away from a process that is actively moving data. In this case, polling simply adds to the staging overhead, slowing down overall throughput. This can be alleviated by the addition of hardware to detect and signal availability of a resource. Using these signals we can suspend a process that has to wait for input data or output buffer space. When the data or space becomes available, the hardware can activate the process. Since process switching time will be reflected in communications overhead, it must be minimized.

As stated earlier, the requirements for computation and communication are roughly the same in algorithms designed for this type of parallel computer. It is also typical for computation and communication to be finely interleaved in these algorithms. In this case, there are no large bursts of data; rather communications activity is constant and is paced

by the computation. Buffering requirements for this situation are minimal.

There are, however, problems for which the interleaving is coarse, that is, computation proceeds over a large data structure for a relatively long time, then some substantial portion of the data structure is moved to another processing node. For example, a matrix inversion algorithm where a subpart of the array is allocated to each processing node would result in this type of behavior. As a result, we will get large bursts of data. A design with fixed-sized buffers will be swamped by this activity. We seek to accommodate these types of problems by providing variable sized buffers.

As the communication coprocessor becomes more sophisticated, the commands from the main processor that direct it become more complex. In contrast, the more independent we make the coprocessor, the less time we have to issue commands. Ideally, the command interface should allow the main processor to completely specify the operations of the coprocessor before the computation begins. There should be no further necessity for guidance during the computation.

In summary, our design goals for a communication coprocessor for the processing elements of a non-shared memory MIMD architecture are:

- Relieve the main processor of communications related activities by eliminating contention and synchronization overhead.
- Reduce the use of intermediate buffering by using computational data structures directly whenever possible.
- Provide a mechanism for mapping logical ports to physical ports.
- Handle multiple physical ports concurrently by using multiple communication processes. Hardware should automatically suspend and activate these processes, and process switching time should be minimized.
- Provide variable sized buffers.
- Provide an efficient command interface that allows the complete specification of the operations of the coprocessor before computation begins.

4 The Processor/Coprocessor Interface

The communication coprocessor must be interfaced to the main processor of a processing node in a way that eliminates communications overhead for the main processor. A first

step toward this is to physically decouple the two processors. This can be done by using a dual-ported memory where the main processor is connected to one port, and the communication coprocessor is connected to the other. Figure 1 shows the basic configuration of a processing node with this type of interface. Data memory contains all data shared by the two processors. The main processor is assumed to have a separate memory for its instructions and private data. The communication port interface is examined in Section 5, while the internal workings of the coprocessor are covered in Section 6. In the remainder of this section we focus on the processor/coprocessor interface.

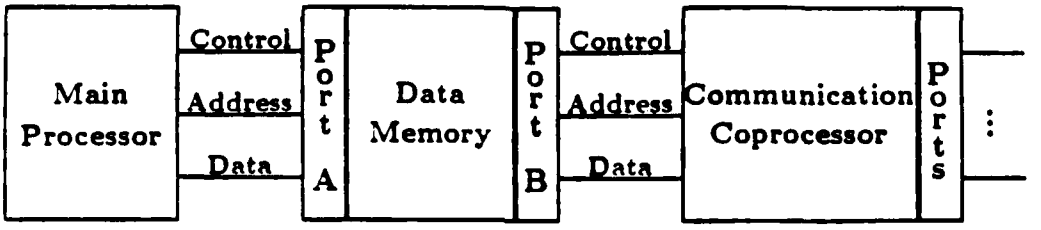


Figure 1: Basic Processor Node Configuration

In the design shown in Figure 1, contention for the shared resource is eliminated by allowing simultaneous access to the data memory. Because we allow simultaneous access, we must address the issue of synchronization. One standard way to do this is to define a buffer area in the memory and use shared pointers to access the buffer. One processor can write to the buffer and the other can read from it. This is sufficient to synchronize data movement between the two processors because each processor modifies only one of the pointers. However, this technique has the overhead of having to compare pointers on every access. Also, this technique does not lend itself to an automatic hardware signalling scheme in case of a full or empty buffer. Rather, these pointers must be polled in software to determine the availability of data or space.

An alternative to this is the synchronization technique used in the HEP multiprocessor [7]. In this architecture, a bit, called the access state, is added to each memory location to indicate whether it is "full" or "empty". A read operation examines the state of a location and when it is "full", it indivisibly reads the data and sets the bit to "empty". Likewise, a write operation waits for an "empty" state, then writes the data and sets the state to "full".

Using this technique in our system eliminates the buffer pointers and the arithmetic operations in the polling sequence. Now, "full" indicates that data is ready to be moved out of a buffer, and "empty" means there is room for data to be placed in a buffer. Two problems remain, however. First, we must have separate input and output buffers. This

is because "full" and "empty" have different meanings for an input buffer and an output buffer. For example, "full" for an input buffer means the main processor can read data, while "full" for an output buffer means that the coprocessor can read. Having separate buffers disambiguates the meaning. The problem with keeping separate buffers is that for a given memory size, the maximum sized problem we can accommodate is reduced by a factor of two. Because of the data memory complexity, it is worth seeking a method that better utilizes this resource.

The method we propose to solve this problem is to add an additional bit to each memory location to indicate whether that location is to be used for "input" or "output". Table 1 gives a summary of the state a location can be in, along with the operation allowed and the state the location is in after an operation. For example, if the location is in the "input-empty" state, it can only be written to by the coprocessor, and after writing, goes to the "input-full" state. Special read/write operations can be defined to bypass this mechanism when necessary. For example, a "peek" operation would look at the value in a buffer without changing the access state.

Initial State		**Operation**	**Final state**	
input	empty	coprocessor can write	input	full
input	full	main processor can read	output	empty
output	empty	main processor can write	output	full
output	full	coprocessor can read	input	empty

Table 1: Memory Location State Table

The second problem that is not solved by using synchronization bits, is that the processors must poll until data is available. As stated previously, this is a potential source of inefficiency because we are using multiple processes. To solve this problem, we must be able to suspend a communication process when a resource is not available, and reactivate it when the resource is available. The scheme to allow this revolves around the buffers in the shared memory. These buffers, called *logical buffers*, connect a computational process in the main processor to a process in the communication coprocessor. The process in the coprocessor in turn connects the logical buffer to a physical port.

Consider processes in the coprocessor first. Because each physical buffer can have many logical buffers associated with it, we may have one process per physical buffer or one process per logical buffer. From the viewpoint of the logical buffers it is best to have a process per logical buffer. This is because the resulting one-to-one correspondence makes it easy to suspend and reactivate processes. For example, an output process is suspended when its associated logical buffer is empty, and reactivated when data is written to the

buffer.

The resulting design for output processes utilizes the logical buffer number to suspend and activate processes. Whenever data is successfully written to a logical output buffer, the buffer number is used to index a process state vector, which keeps track of whether the process is suspended or active. If the process is already active, then nothing happens. If the process is suspended, this was a write to an empty buffer, and the process state is set to active. When an output process tries to read an empty buffer, the failed read sets the process state to suspended. On the physical buffer side, an output process is suspended when it tries to write to a full buffer. When space becomes available in the physical buffer, *all* processes waiting for the buffer should be reactivated. This is accomplished with a global associative map with an entry for each output process. The number of the physical port now ready is applied to this map, which generates an activation signal for each process associated with the port. All physical ports must use this table, so a priority encoder is used to control access to it.

On the input side, associating one process per logical buffer does not work, because the mapping is one-to-many; there is one data item at the head of the input queue, which is destined for one of many logical buffers. Multiple processes would simply delay this data transfer. Alternatively, consider having one process for each physical input buffer. Now data in the input buffer is transferred to its destination logical port without delay. This process will be suspended if either the physical buffer is empty, or the current destination logical buffer is full. If the process is suspended because the physical buffer is empty, then the process is simply reactivated whenever data arrives in this buffer. For a process suspended because a logical buffer is full, we must provide a mechanism to signal a successful read of that particular logical buffer. This can be accomplished with an associative map with one entry for each physical input buffer. The logical buffer number that caused the process to be suspended is stored in this table. When the main processor successfully reads a logical buffer, the number of the logical buffer is passed through this table. If an entry matches the logical buffer number, the corresponding process is activated.

Now consider the main processor. In this case, a process can have a number of ports, some input and some output. We assume that this number is fixed, and that the number of input ports equals the number of output ports. In addition, we assume that processes in the main processor perform blocking reads and writes. This means that a process will be suspended if an input port is empty or an output port is full. A lookup table with one entry per process is sufficient to keep track of this. All logical buffer accesses that succeed are passed through this table to activate processes.

Before we proceed to the command interface, let us summarize the design to this point as it relates to our goals. The two processors have been effectively isolated by a dual-ported memory. Synchronization bits allow this memory to be shared, and give

the coprocessor an efficient means of accessing the data structures of the main processor. This architecture allows variable sized buffers and logical-to-physical port mapping to be handled in a straight-forward way. In addition, large bursts of data are easily handled because data is placed directly in the destination data structure. A complete hardware signalling structure eliminates all polling.

The command interface between the main processor and the coprocessor must allow the main processor to send the logical-to-physical port mapping, and the size and location of each logical buffer to the coprocessor. Since communication between the processors is through the shared memory, these commands must be sent using this facility. The most straight-forward way to do this is to define a logical buffer for the communication of commands, and to have an initialization process in each processor that communicate with this buffer. If we allow two-way communications between these processes, then the coprocessor can send status information back to the main processor.

5 The Communication Port Interface

The communication coprocessor moves data between the interface described in Section 4 and a number of input and output ports. These ports are grouped into input/output pairs to form channels, which can be used to connect processing nodes together. All channels are identical, operate in parallel, and perform all of the operations involved in transmitting data from a port buffer to another processor.

A block diagram of the hardware associated with a channel is shown in Figure 2. Each port consists of a data buffer, a buffer controller, a register to hold the current logical port number, and logic to process the data. In the input port, this logic converts incoming data to its internal form, detects protocol bits, and detects errors. This logic also detects a change in the logical port number, updating the register appropriately. The output port logic converts data to its transmission form, generates protocol and error detection codes, and detects changes in the logical port number.

The port controllers orchestrate the movement of data within the channel, and generate signals to the coprocessor's main control section by consulting status information from the buffer controller. When an attention signal is generated, the cause is passed to the coprocessor control where it is used to control the execution of the process associated with the port. For example, when the input buffer goes from empty to almost empty, its process is activated, and when the buffer becomes empty, the process is suspended. Table 2 summarizes the meanings of the port status bits.

An integral part of the design of the channel hardware, is the design of the packet and

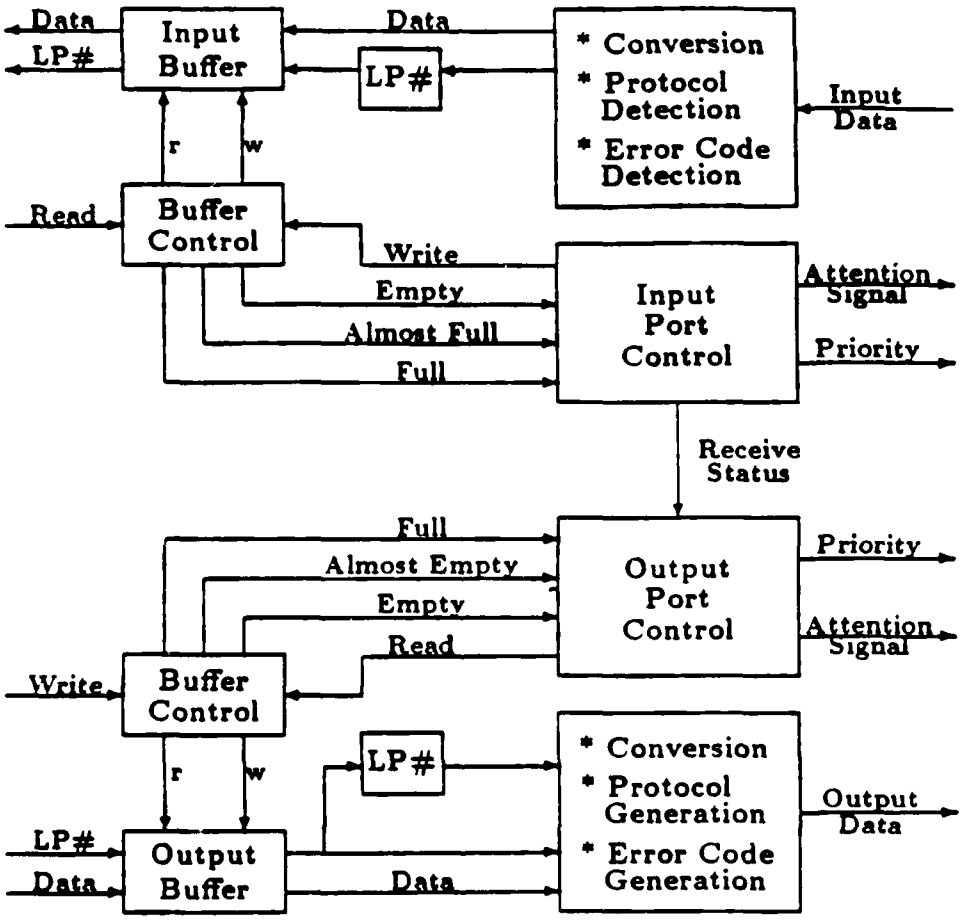


Figure 2: Channel Architecture (LP# = Logical port number)

Input	
Status	Action
empty	suspend process
not empty	activate process at low priority
almost full	medium priority
full	hold sender, high priority

Output	
Status	Action
empty	high priority
almost empty	medium priority
full	suspend process
not full	activate process at low priority

Table 2: Buffer Status Information

transmission protocol. In general, a packet consists of a header followed by a packet body which may contain a logical port number, data, and error detection information. Within this structure, four types of packets are allowed. The second two bits of the packet indicate the packet type as shown in Table 3. The first bit is a start bit. When a packet arrives, its type is determined after the first three bits. If it is a data packet, a request is issued to the output port to send an acknowledgment, and if there is a logical port number, it is extracted. If a subsequent error is detected in the packet, a request for retransmission is sent. Since the acknowledge has already been sent, the sender is transmitting the next packet when it receives the retransmit request. To prevent overwriting this new packet, the acknowledge for the retransmitted packet will not be sent until it has been completely processed.

Header	Packet Body	Function
100	empty	acknowledge
101	empty	retransmit request
110	logical port, data, error code	data transfer; set logical port
111	data and error codes	data transfer

Table 3: Allowed Packet Formats

The basic unit of data to be transmitted via a channel is a 32-bit word. The architecture will be tuned to this unit, so sending single bytes or half-words will be expensive, but will

almost never be required. It will be more common to have four bytes or two half-words packed together. The number of bits of error code accompanying the 32 bits of data will depend on the transmission error rate. The higher the error rate, the more bits will be needed for detection. Some possible error codes and their impact on transmission overhead are shown in Table 4. Note that error correction codes are not a viable alternative for the expected error rates. For example, compare a seven-bit error correcting code to the 4-bit parity scheme. An error rate greater than 1 in 15 packets would be required to justify the error correction code. We expect the error rates to be very much less than this.

Error Code	Overhead
7-bit ECC	29%
Byte Parity	24%
Word Parity	18%

Table 4: Error Codes and their Overhead

In summary, the port interface design presented above reduces transmission overhead by allowing all channels to operate in parallel, and by having the channel perform all transmission related operations. Also, the protocol presented provides flexibility while minimizing transmission overhead.

6 Overall Design

A high level block diagram of the overall architecture of the communication coprocessor is shown in Figure 3. As shown, channels are connected together by a data bus and a control bus. The coprocessor controller unit handles the processes which move data between the port buffers and the external dual-ported memory, and handles the logical-to-physical mapping of buffers described in Section 5. Control signals to and from the data memory include the operation code (read, write, peek, etc.), the logical port number, and operation status signals.

The mechanism used in the coprocessor control unit to handle communication processes is shown in Figure 4. Signals from the mapping tables and data memory, are applied to the process status vector, which holds the state of each process. A process can be either suspended or active, so each element of the status vector has one bit.

The status vector is used to control the token ring by allowing active processes to hold the token and by causing suspended processes to be passed over when the token moves.

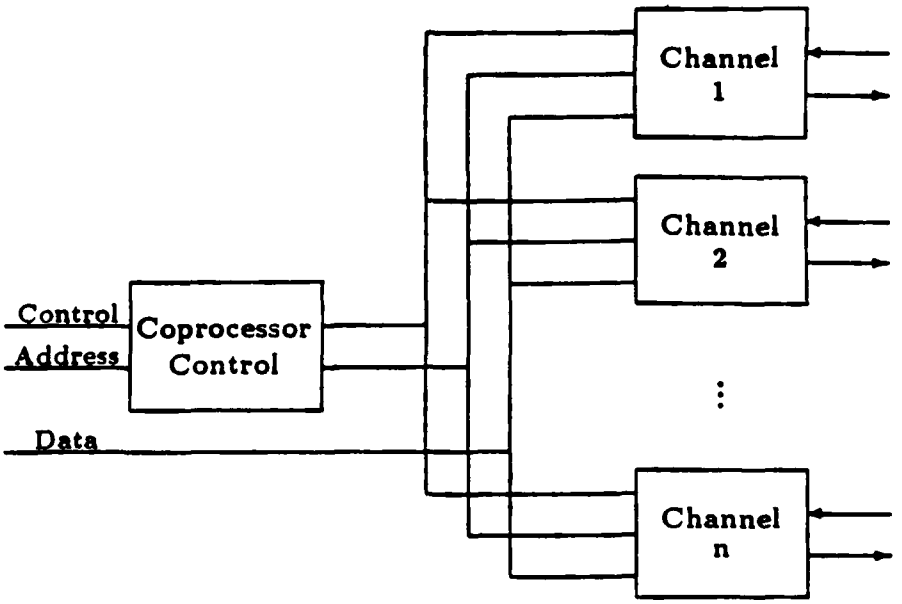


Figure 3: High Level Block Diagram of the Communication Coprocessor

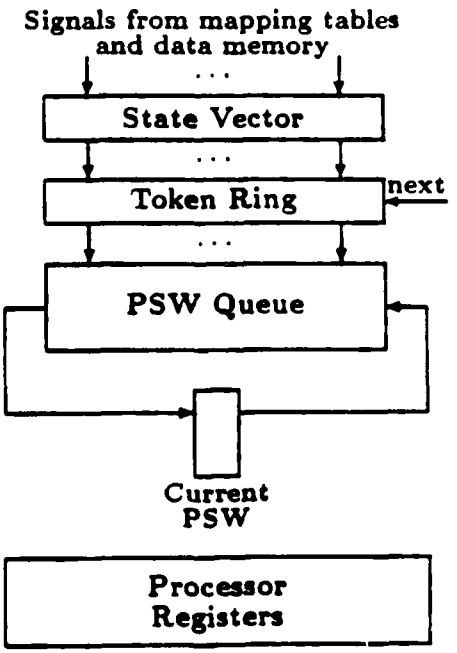


Figure 4: Mechanism for Handling Multiple Processes

The position of the token, in turn, indicates the process currently being executed. When the "next" signal is applied to the token ring, the token moves to the next active process. A constraint on the token ring is that it must be able to move the token through the entire ring in one process execution cycle. This is necessary for the case of one active process.

The process queue has one entry for each output process and one entry for each input process. The size of this queue is then the sum of the maximum number of logical output buffers and the number of physical input buffers. Alongside this queue is a memory that contains the values defining each logical buffer. Five values are required to define each logical buffer: the starting location, the ending location, the head pointer, the tail pointer and the step size to get to the next element. Logical output process numbers can be used to index this table directly to obtain the correct set of values. For input processes we must concatenate the process number and the logical buffer number to get an index.

One advantage of this process control mechanism is that addition and deletion to and from the "logical" process queue, which is variable sized, has no impact on the performance of the coprocessor. Setting the state vector entry for a process to suspended automatically removes it from the queue, and setting the state to active automatically adds it to the queue. Another advantage is that the process switch time is zero. This is due in part to the token ring, and in part to the separate storage allocated to each process.

When a process runs, it can execute any number of cycles before the control unit switches to the next process. However, to prevent any individual buffer from filling up, we want to cycle through all processes in as short a time as possible. The smallest execution step for a process is the movement of one unit of data between a physical port and a logical port. The actual number of these steps executed will depend on the process priority, which is set by the state of the physical buffer (see Section 5), with the lowest priority processes executing one step.

To better understand how this design reduces communications overhead, consider an example illustrating how data would move between data structures in two different processors. To begin, the main processor writes data to a data structure, then goes on with its computations. The location written to is now part of a logical output buffer. Suppose this buffer was empty. This means that the corresponding communication process is suspended and that the write to the empty buffer will activate this process. When this process reaches the head of the process queue, the coprocessor will move the data from the logical buffer to the physical buffer in the port. The data is then transmitted to the destination port buffer automatically and independent of the coprocessor. When the data arrives at its destination port buffer, it will then be moved to a logical port by the process associated with the physical port.

Now consider the staging overhead in the data transfer just described. Data availability

is signalled automatically as soon as the data is written to the logical buffer. There is no overhead in this operation for either the sender or the receiver. Data movement has been minimized to one memory read and one memory write at each end of the transfer. Finally, preparation of the data packet for transmission is done automatically be special channel hardware. This overhead as well as the transmission overhead are completely hidden by permitting data movement within the coprocessor to proceed in parallel.

7 Conclusions

Non-shared memory MIMD architectures must provide efficient interprocessor communication in order to meet the demands of the algorithms developed for them. Existing architectures do not provide the necessary communication throughput, hence they have a "communications bottleneck" that can severely limit performance. The sources of this "bottleneck" are staging overhead, transmission overhead, and bandwidth limitations.

We have addressed the problem of efficient interprocessor communications in this type of MIMD architecture by introducing a communication coprocessor that minimizes staging and transmission overhead and relieves the main processor of all communications related activities. The coprocessor is multiprocessing with zero process switch time, allowing it to efficiently handle a number of communication channels. In addition, each channel operates independently of and in parallel with the other channels, with transmission and reception of data performed simultaneously within a channel. The protocol utilized minimizes transmission overhead while providing flexibility and fault tolerance.

References

[1] *INMOS Databook*. INMOS Corporation, 1986.

[2] C. Lutz, S. Rabin, C. Seitz, and D. Speck. Design of the Mosaic Element. In *Proceedings, Conference on Advanced Research in VLSI*, pages 1-10, Artech, 1984.

[3] J.C. Peterson, J.O. Tuazon, D. Lieberman, and M. Pneil. The Mark III Hypercube-Ensemble Concurrent Computer. In *Proceedings of the International Conference on Parallel Processing*, pages 71-73, IEEE, 1986.

[4] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247-1265, 1984.

[5] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28:22-33, Jan 1985.

[6] Charles L. Seitz. Ensemble Architechures for VLSI - A Survey and Taxonomy. In Paul Penfield, editor, *Proceedings, Conference on Advanced Research in VLSI*, pages 130–135, 1982.

[7] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Proceedings of SPIE - The International Society for Optical Engineering*, 298:241–248, 1981.

[8] Lawrence Snyder. Introduction to the Configurable, Highly Parallel Computer. *Computer*, 15(1):47–56, 1982.

[9] J. Tuazon, J. Peterson, M. Pneil, and D. Lieberman. Caltech/JPL Mark II Hypercube Concurrent Processor. In *Proceedings of the International Conference on Parallel Processing*, pages 666–673, IEEE, 1986.

END

DATE

FILMED

DTIC

JULY 88